SUPPORT DU JAVA AVANCE

RMI, Sérialisation, JDBC, JNI

Encadré par M.BOULCHAHOUB

Les chapitres contenus dans ce support :

Chapitre 1 : L'appel de méthodes distantes : RMI

RMI (Remote Method Invocation) est une technologie développée et fournie par Sun à partir du JDK 1.1 pour permettre de mettre en œuvre facilement des objets distribués.

Ce chapitre contient plusieurs sections :

- Présentation et architecture de RMI
- Les différentes étapes pour créer un objet distant et l'appeler avec RMI
- Le développement coté serveur
- Le développement coté client
- La génération des classes stub et Skelton
- La mise en œuvre des objets RMI
- Les TPs de manipulation l'API RMI

1.1. Présentation et architecture de RMI

Le but de RMI est de permettre l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine virtuelle différente de celle de l'objet l'appelant. Cette machine virtuelle peut être sur une machine différente pourvu qu'elle soit accessible par le réseau.

La machine sur laquelle s'exécute la méthode distante est appelée serveur.

L'appel coté client d'une telle méthode est un peu plus compliqué que l'appel d'une méthode d'un objet local mais il reste simple. Il consiste à obtenir une référence sur l'objet distant puis à simplement appeler la méthode à partir de cette référence.

La technologie RMI se charge de rendre transparente la localisation de l'objet distant, son appel et le renvoi du résultat. En fait, elle utilise deux classes particulières, le stub et le Skelton, qui doivent être générées avec les outils fournis avec le JDK.

Le stub est une classe qui se situe côté client et le skeleton est son homologue coté serveur. Ces deux classes se chargent d'assurer tous les mécanismes d'appel, de communication, d'exécution, de renvoie et de réception du résultat.

1.2. Les différentes étapes pour créer un objet distant et l'appeler avec RMI

Le développement coté serveur se compose de :

- La définition d'une interface qui contient les méthodes qui peuvent être appelées à distance
- L'écriture d'une classe qui implémente cette interface
- L'écriture d'une classe qui instanciera l'objet et l'enregistrera en lui affectant un nom dans le registre de nom RMI (RMI Registry)

Le développement côté client se compose de :

- L'obtention d'une référence sur l'objet distant à partir de son nom
- L'appel à la méthode à partir de cette référence

1.3. Le développement coté serveur

1.3.1. La définition d'une interface qui contient les méthodes de l'objet distant

L'interface à définir doit hériter de l'interface java.rmi.Remote. Cette interface ne contient aucune méthode mais indique simplement que l'interface peut être appelée à distance.

L'interface doit contenir toutes les méthodes qui seront susceptibles d'être appelées à distance.

La communication entre le client et le serveur lors de l'invocation de la méthode distante peut échouer pour diverses raisons comme un crash du serveur, une rupture de la liaison, etc …

Ainsi chaque méthode appelée à distance doit déclarer qu'elle est en mesure de lever l'exception java.rmi.RemoteException.

```
package test_rmi;

import java.rmi.*;

public interface Information extends Remote {

   public String getInformation() throws RemoteException;

}
```

1.3.2. L'écriture d'une classe qui implémente cette interface

Cette classe correspond à l'objet distant. Elle doit donc implémenter l'interface définie et contenir le code nécessaire.

Cette classe doit obligatoirement hériter de la classe UnicastRemoteObject qui contient les differents traitements élémentaires pour un objet distant dont l'appel par le stub du client est unique. Le stub ne peut obtenir qu'une seule référence sur un objet distant héritant de UnicastRemoteObject. On peut supposer qu'une future version de RMI sera Capable de faire du MultiCast, permettant à RMI de choisir parmi plusieurs objets distants identiques la référence à fournir au client.

La hiérarchie de la classe UnicastRemoteObject est :

```
java.lang.Object
        java.rmi.Server.RemoteObject
            java.rmi.Server.RemoteServer
                    java.rmi.Server.UnicastRemoteObject
```

Comme indiqué dans l'interface, toutes les méthodes distantes doivent indiquer qu'elles peuvent lever l'exception RemoteException mais aussi le constructeur de la classe. Ainsi, même si le constructeur ne contient pas de code il doit être redéfini pour inhiber la génération du constructeur par défaut qui ne lève pas cette exception.

```
package test_rmi;

import java.rmi.*;

import java.rmi.server.*;

public class TestRMIServer extends UnicastRemoteObject implements Information {

   protected TestRMIServer() throws RemoteException {
      super();
   }

   public String getInformation()throws RemoteException {
      return "bonjour";
   }

}
```

1.3.3. L'écriture d'une classe pour instancier l'objet et l'enregistrer dans le registre

Ces opérations peuvent être effectuées dans la méthode main d'une classe dédiée ou dans la méthode main de la classe de l'objet distant. L'intérêt d'une classe dédié et qu'elle permet de regrouper toutes ces opérations pour un ensemble d'objets distants.
La marche a suivre contient trois étapes :

- La mise en place d'un Security manager dédié qui est facultative
- L'instanciation d'un objet de la classe distante
- L'enregistrement de la classe dans le registre de nom RMI en lui donnant un nom

1.3.4. L'enregistrement dans le registre de nom RMI en lui donnant un nom

La dernière opération consiste à enregistrer l'objet créé dans le registre de nom en lui affectant un nom. Ce nom est fourni au registre sous forme d'une URL constitué du préfix rmi://, du nom du seveur (hostname) et du nom associé à l'objet précédé d'un slash.

Le nom du serveur peut être fourni « en dur » sous forme d'une constante chaine de caractères ou peut être dynamiquement obtenu en utilisant la classe InetAddress pour une utilisation en locale.

C'est ce nom qui sera utilisé dans une URL par le client pour obtenir une référence sur l'objet distant.

L'enregistrement se fait en utilisant la méthode rebind de la classe Naming. Elle attend en paramètre l'URL du nom de l'objet et l'objet lui même.

```
public static void main(String[] args) {

   try {

      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());

      TestRMIServer testRMIServer = new TestRMIServer();

      System.out.println("Enregistrement du serveur");

      Naming.rebind("rmi://"+java.net.InetAddress.getLocalHost()+
         "/TestRMI",testRMIServer);

      // Naming.rebind(";rmi://localhost/TestRMI", testRMIServer);

      System.out.println("Serveur lancé");

   } catch (Exception e) {
      System.out.println("Exception capturée: " + e.getMessage());
   }

}
```

1.3.5. Lancement dynamique du registre de nom RMI

Sur le serveur, le registre de nom RMI doit s'exécuter avant de pouvoir enregistrer un objet ou obtenir une référence.

Ce registre peut être lancé en tant qu'application fournie par sun dans le JDK (rmiregistry) ou peut être lancé dynamiquement dans la classe qui enregistre l'objet. Ce lancement ne doit avoir lieu qu'une seule et unique fois. Il peut être intéressant d'utiliser ce code si l'on créé une classe dédié à l'enregistrement des objets distants.

Le code pour exécuter le registre est la méthode createRegistry de la classe java.rmi.registry.LocateRegistry. Cette méthode attend en paramètre un numéro de port.

```
public static void main(String[] args) {

   try {

      java.rmi.registry.LocateRegistry.createRegistry(1099);

      System.out.println("Mise en place du Security Manager ...");
      System.setSecurityManager(new java.rmi.RMISecurityManager());

      ...

   }
}
```

1.4. Le développement coté client

1.4.1. La mise en place d'un Security Manager

Comme pour le coté serveur, cette opération est facultative.
Le choix de la mise en place d'un Security manager côté client suit des règles identiques à celui du côté serveur. Sans son utilisation, il est nécessaire de mettre dans le CLASSPATH du client toutes les classes nécessaires dont la classe stub.

```
public static void main(String[] args) {

   System.setSecurityManager(new RMISecurityManager());

}
```

1.4.2. L'obtention d'une référence sur l'objet distant à partir de son nom

Pour obtenir une référence sur l'objet distant à partir de son nom, il faut utiliser la méthode statique lookup() de la classe Naming.

Cette méthode attend en paramètre une URL indiquant le nom qui référence l'objet distant. Cette URL est composé de préfix rmi://, le nom du serveur (hostname) et le nom de l'objet tel qu'il a été enregistré dans le registre précédé d'un slash.

Il est préférable de prévoir le nom du serveur sous forme de paramètres de l'application ou de l'applet pour plus de souplesse.

La méthode lookup() va rechercher dans le registre du serveur l'objet et retourner un objet stub. L'objet retourné est de la classe Remote (cette classe est la classe mère de tous les objets distants).

Si le nom fourni dans l'URL n'est pas référencé dans le registre, la méthode lève l'exception NotBoundException.

```
public static void main(String[] args) {

   System.setSecurityManager(new RMISecurityManager());

   try {

      Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

   } catch (Exception e) {
   }

}
```

1.4.3. L'appel à la méthode à partir de la référence sur l'objet distant

L'objet retourné étant de type Remote, il faut réaliser un cast vers l'interface qui définit les méthodes de l'objet distant.

Pour plus de sécurité, on vérifie que l'objet retourné est bien une instance de cette interface. Un fois le cast réalisé, il suffit simplement d'appeler la méthode.

```
public static void main(String[] args) {

   System.setSecurityManager(new RMISecurityManager());

   try {

      Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

      if (r instanceof Information) {
         String s = ((Information) r).getInformation();
         System.out.println("chaine renvoyée = " + s);
      }

   } catch (Exception e) {
   }
}
```

1.4.4. L'appel d'une méthode distante dans une applet

L'appel d'une méthode distante est la même dans une application et dans une applet. Seule la mise en place d'un Security Manager dédié dans les applets est inutile car elles utilisent déjà un Security Manager (AppletSecurityManager) qui autorise le chargement de classes distantes.

```
public void init() {

   try {
      Remote r = Naming.lookup("rmi://vaio/127.0.0.1/TestRMI");

      if (r instanceof Information) {
         s = ((Information) r).getInformation();
      }
   } catch (Exception e) {
   }

}
```

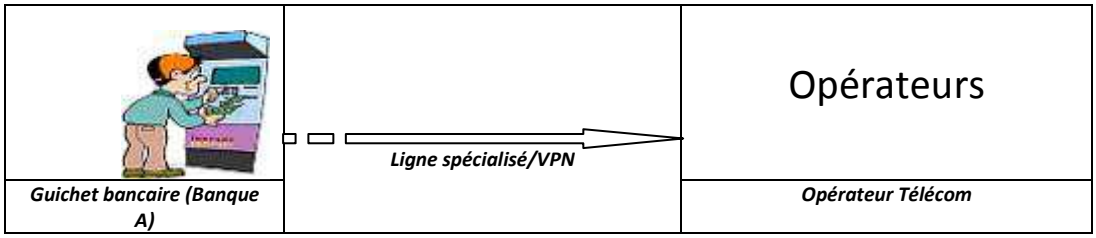
1.5. La mise en œuvre des objets RMI

La mise en œuvre et l'utilisation d'objet distant avec RMI nécessite plusieurs étapes :

1. Démarrer le registre RMI sur le serveur soit en utilisant le programme rmiregistry livré avec le JDK soit en exécutant une classe qui effectue le lancement.
2. Exécuter la classe qui instancie l'objet distant et l'enregistre dans le serveur de nom RMI
3. Lancer l'application ou l'applet pour tester.

1.6. Les TP de manipulation de l'API RMI

Dans cet atelier, nous traiterons le cas de la recharge d'un téléphone mobile à partir du guichet bancaire automatique.



Pour recharger son téléphone mobile à partir d'un guichet bancaire, le client doit saisir le numéro de téléphone et le montant de la recharge à utiliser.

Nous aurons besoin de créer deux classes

Supposons alors qu'une facture est définie par son numéro et son client.

1.6.1 Le fournisseur du service de recharge

Les éléments à implémenter et les étapes à mettre en œuvre dans la plateforme du fournisseur sont détaillés comme suivant :

1. Créer un projet « II-OperateurTelecom » et créer le package modèle « ma.operateur.telecom.model »
2. Dans le package modèle « ma.operateur.telecom.model », créer deux classes Client et Recharge.
3. La classe client est défini par l'attribut numTelephone de type chaine de caractères. Générer automatiquement les Getters et les Setters de cet attribut. Surcharger le constructeur de cette classe automatiquement.

```
public class Client {

	private String numTelephone;

	public String getNumTelephone() {

		return numTelephone;

	}

	public void setNumTelephone(String numTelephone) {

		this.numTelephone = numTelephone;

	}

	public Client(String numTelephone) {
```

```
                super();
                this.numTelephone = numTelephone;
        }
}
```

4. La classe Recharge est définie par l'attribut clientRecharge de type Client, et montantRecharge de type double. Générer automatiquement les Getters et les Setters de ces attributs. Surcharger le constructeur de cette classe automatiquement.

```
public class Recharge {
        private Client clientRecharge;
        private double montantRecharge;
        public Recharge(Client clientRecharge, double montantRecharge) {
                super();
                this.clientRecharge = clientRecharge;
                this.montantRecharge = montantRecharge;
        }
        public Client getClientRecharge() {
                return clientRecharge;
        }
        public void setClientRecharge(Client clientRecharge) {
                this.clientRecharge = clientRecharge;
        }
        public double getMontantRecharge() {
                return montantRecharge;
        }
        public void setMontantRecharge(double montantRecharge) {
                this.montantRecharge = montantRecharge;
        }
}
```

5. Maintenant, nous allons créer un package pour les services à exposer par l'opérateur Télécom. Appeler le « ma.operateur.telecom.services »
6. Dans le package « ma.operateur.telecom.services », créer une interface IOperateurTelecomServices.

Cette interface sera considérée comme un contrat entre le fournisseur (L'opérateur Télécom) et le client (La banque) de notre service de recharge.

7. Dans le contrat crée, mettre la signature de la méthode qui permet de recharger le solde. Cette méthode prend en paramètre un objet de type Recharge et retourne un boolean.

```
package ma.operateur.telecom.services;

import ma.operateur.telecom.model.Recharge;

public interface IOperateurTelecomServices {

	public boolean rechargerSolde(Recharge recharge);

}
```

8. Le contrat crée, doit respecter les règles exigées par le RMI.

- *Le contrat doit étendre l'interface java.rmi.Remote*
- *Les services exposés en RMI doivent remonter des exceptions de types java.rmi.RemoteException*

9. Ajouter le lien d'héritage avec java.rmi.Remote et remonter les exceptions java.rmi.RemoteException.

```
package ma.operateur.telecom.services;

import java.rmi.Remote;

import java.rmi.RemoteException;

import ma.operateur.telecom.model.Recharge;

public interface IOperateurTelecomServices  extends Remote{

public boolean rechargerSolde(Recharge recharge) throws
RemoteException;

}
```

10. Maintenant créer la classe OperateurTelecomServicesImpl qui implémente l'interface IOperateurTelecomServices dans le package « ma.operateur.telecom.services ». Dans l'implémentation de la méthode rechargeSolde, faire une trace pour s'assurer de l'exécution de cette méthode lors de l'invocation d'une banque.

```
package ma.operateur.telecom.services;

import java.rmi.RemoteException;

import ma.operateur.telecom.model.Recharge;

public class OperateurTelecomServicesImpl implements
IOperateurTelecomServices{

	@Override
```

```
	public boolean rechargerSolde(Recharge recharge) throws
RemoteException {

		System.out.println("Une banque a fait l'invocation de cette
méthode");

		return false;

	}

}
```

11. La classe qui implémente le contrat, doit respecter les règles exigées par le RMI.

- *L'implémentation du contrat RMI doit étendre la classe java.rmi.server.UnicastObjectRemote*

12. Ajouter le lien d'héritage entre OperateurTelecomServicesImpl et UnicastRemoteObjet. Une erreur de compilation apparaîtra alors dans votre programme et elle est justifiée par le fait que le constructeur de la classe UnicastRemoteObjet remonte des exceptions de type RemoteException. Dans le constructeur de la classe OperateurTelecomServicesImpl Remonter également des exceptions de type RemoteException pour corriger cette erreur de compilation.

```
package ma.operateur.telecom.services;

import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

import ma.operateur.telecom.model.Recharge;

public class OperateurTelecomServicesImpl extends UnicastRemoteObject
implements IOperateurTelecomServices{



	protected OperateurTelecomServicesImpl() throws RemoteException {

		super();

		// TODO Auto-generated constructor stub

	}

	@Override

	public boolean rechargerSolde(Recharge recharge) throws
RemoteException {

		System.out.println("Une banque a fait l'invocation de cette
méthode");

		return false;

	}

}
```

13. Après avoir implémenté les trois règles exigées par le RMI, nous allons créer un serveur RMI qui va attendre les demandes qui vont parvenir des banques. Il s'agit d'une classe java contenant la méthode main. Elle contiendra les trois instructions suivantes :
    a. Réserver un port RMI pour être à l'écoute.

```
LocateRegistry.createRegistry(1099);
```

    b. Instancier à place de la banque un objet de la classe OperateurTelecomServicesImpl. La banque ne peut pas instancier cet objet car elle n'aura pas la classe OperateurTelecomServicesImpl qui contient le code de la méthode de recharge (c'est une méthode hyper sensible)

```
IOperateurTelecomServices contrat = new
OperateurTelecomServicesImpl();
```

    c. Associer à l'objet crée dans le point (b) une URL RMI.

```
Naming.bind("rmi://localhost:1099/test", contrat);
```

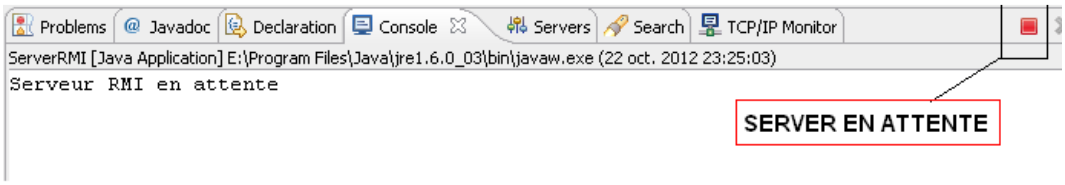
14. Dans le package « ma.operateur.telecom.services », créer une classe ServerRMI contenant la méthode main avec les trois instructions cités dans le point précédent.

```
package ma.operateur.telecom.services;

import java.net.MalformedURLException;

import java.rmi.AlreadyBoundException;

import java.rmi.Naming;

import java.rmi.RemoteException;

import java.rmi.registry.LocateRegistry;

public class ServerRMI {

   /**

    * @param args

    */

   public static void main(String[] args) {

         try {

LocateRegistry.createRegistry(1099);

               IOperateurTelecomServices contrat = new
OperateurTelecomServicesImpl();

               Naming.bind("rmi://localhost:1099/test", contrat);

               System.out.println("Serveur RMI en attente");
```

```
            } catch (RemoteException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (MalformedURLException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            } catch (AlreadyBoundException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
}
```

- *L'URL RMI doit respecter la forme exigée par le RMI suivante* ***rmi://server :port/nomObjet***

15. Faire l'exécution de la classe ServerRMI et s'assurer qu'elle reste en attente.

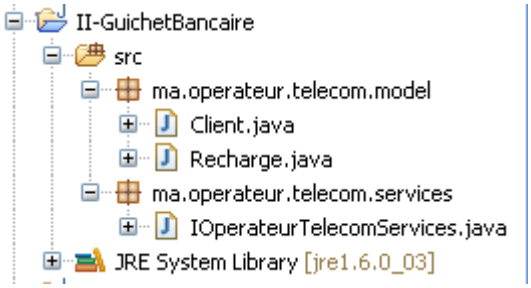


1.6.2 Le client du service de recharge (La banque)

Le fournisseur d'un service doit donner à son client les classes modèles (les classe Client et Recharge) et le contrat du service (l'interface IOperateurTelecomServices).

- *Le client d'un service doit disposer des classes modèles et du contrat du service*

16. Créer un nouveau projet et appeler le « II-GuichetBancaire ». Copier dans ce projet les classes modèles et le contrat à partir du projet « II-OperateurTelecom ». Il faut respecter les mêmes packages



17. Créer ensuite un package « ma.gb.presentation » dans le projet « II-GuichetBancaire ». Puis une classe InvoquerServicesOperateur contenant la méthode main.
18. Dans la méthode main : de la classe InvoquerServicesOperateur, rechercher l'objet des services auquel le fournisseur a attribué l'url RMI suivante : rmi://localhost :1099/test

```
IOperateurTelecomServices contrat = (IOperateurTelecomServices)
Naming.lookup("rmi://localhost:1099/test");
```

19. Dans la méthode main : Instancier un objet de type client puis un objet de type Recharge.

```
   Client clientRecharge = new Client("0660980989");

Recharge recharge = new Recharge(clientRecharge, 50.00d);
```

20. Dans la méthode main : Invoquer la méthode rechargerSolde en passant l'objet recharge comme argument. Tracer ensuite le retour du service de recharge

```
boolean resultatRecharge = contrat.rechargerSolde(recharge);
System.out.println(resultatRecharge);
```

21. Dans la méthode main : Invoquer la méthode rechargerSolde en passant l'objet recharge comme argument. Tracer ensuite le retour du service de recharge

```
boolean resultatRecharge = contrat.rechargerSolde(recharge);
System.out.println(resultatRecharge);
```

22. Faire l'exécution de la classe InvoquerServicesOperateur et remarquer l'exception par rapport aux classes Client et Recharge qui ne sont pas sérialisables.

```
java.rmi.MarshalException: error marshalling arguments; nested exception is:

    java.io.NotSerializableException: ma.operateur.telecom.model.Recharge

    at sun.rmi.server.UnicastRef.invoke(Unknown Source)

    at java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod(Unknown Source)

    at java.rmi.server.RemoteObjectInvocationHandler.invoke(Unknown Source)

    at $Proxy0.rechargerSolde(Unknown Source)

    at ma.gb.presentation.InvoquerServicesOperateur.main(InvoquerServicesOperateur.java:20)

Caused by: java.io.NotSerializableException: ma.operateur.telecom.model.Recharge

    at java.io.ObjectOutputStream.writeObject0(Unknown Source)
```

```
    at java.io.ObjectOutputStream.writeObject(Unknown Source)

    at sun.rmi.server.UnicastRef.marshalValue(Unknown Source)

    ... 5 more
```

23. Ajouter l'implémentation de java.io.Serializable pour les classes Client et Recharge côté fournisseur et côté client. Redémarrer le serveur RMI et essayer d'invoquer le service de recharge à nouveau

- *Le client d'un service ne doit jamais avoir accès à la classe d'implémentation. Dans notre cas la banque ne devra jamais avoir la classe* `OperateurTelecomServicesImpl`

Chapitre 2 : La sérialisation binaire des objets

La sérialisation est un procédé introduit dans le JDK version 1.1 qui permet de rendre un objet persistant. Cet objet est mis sous une forme sous laquelle il pourra être reconstitué à l'identique. Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer dans une autre JVM. C'est le procédé qui est utilisé par RMI. La sérialisation est aussi utilisée par les beans pour sauvegarder leurs états.
Au travers de ce mécanisme, java fourni une façon facile, transparente et standard de réaliser cette opération : ceci permet de facilement mettre en place un mécanisme de persistance. Il est de ce fait inutile de créer un format particulier pour sauvegarder et relire un objet. Le format utilisé est indépendant du système d'exploitation. Ainsi, un objet sérialisé sur un système peut être réutilisé par un autre système pour récréer l'objet.
L'ajout d'un attribut à l'objet est automatiquement pris en compte lors de la sérialisation. Attention toutefois, la déserialisation de l'objet doit se faire avec la classe qui à été utilisée pour la sérialisation. La sérialisation peut s'appliquer facilement à tous les objets.

Ce chapitre présente dans plusieurs sections l'utilisation de cette API

2.1. Les classes et les interfaces de la sérialisation

La sérialisation définie l'interface Serializable et les classes ObjectOutputStream et ObjectInputStream

2.1.1. L'interface Serializable

Cette interface ne définie aucune méthode mais permet simplement de marquer une classe comme pouvant être sérialisée.

Tout objet qui doit être sérialiser doit implémenter cette interface ou une de ses classes mères doit l'implémenter.

Si l'on tente de sérialiser un objet qui n'implémente pas l'interface Serializable, une exception NotSerializableException est levée.

```
public class Personne implements java.io.Serializable {
private String nom = "";
private String prenom = "";
private int taille = 0;
public Personne(String nom, String prenom, int taille) {
this.nom = nom;
this.taille = taille;
this.prenom = prenom;
}
public String getNom() {
return nom;
}
public void setNom(String nom) {
this.nom = nom;
}
public int getTaille() {
return taille;
}
public void setTaille(int taille) {
this.taille = taille;
}
public String getPrenom() {
return prenom;
}
public void setPrenom(String prenom) {
this.prenom = prenom;
}
}
```

2.1.2. La classe ObjectOuputStream

Cette classe permet de sérialiser un objet.

```
import java.io.*;
public class SerializerPersonne {
public static void main(String argv[]) {
Personne personne = new Personne("Dupond","Jean",175);
try {

FileOutputStream fichier = new
FileOutputStream("personne.ser");
ObjectOutputStream oos = new ObjectOutputStream(fichier);
oos.writeObject(personne);
oos.flush();
oos.close();

}
catch (java.io.IOException e) {
e.printStackTrace();
}
}
}
```

On définit un fichier avec la classe FileOutputStream. On instancie un objet de classe ObjectOutputStream en lui fournissant en paramètre le fichier : ainsi, le résultat de la sérialisation sera envoyé dans le fichier.

On appel la méthode writeObject en lui passant en paramètre l'objet à sérialiser. On appelle la méthode flush() pour vider le tampon dans le fichier et la méthode close() pour terminer l'opération.

Lors de ces opérations une exception de type IOException peut être levée si un problème intervient avec le fichier.

Après l'exécution de cet exemple, un fichier nommé « personne.ser » est créé. On peut visualiser sont contenu mais surtout pas le modifier car sinon il serait corrompu. En effet, les données contenues dans ce fichier ne sont pas toutes au format caractères.

La classe ObjectOutputStream contient aussi plusieurs méthodes qui permettent de sérialiser des types élémentaires et non des objets : writeInt, writeDouble, writeFloat ...
Il est possible dans un même flux d'écrire plusieurs objets les uns à la suite des autres. Ainsi plusieurs objets peuvent être sauvegardés. Dans ce cas, il faut faire attention de relire les objets dans leur ordre d'écriture.

2.1.3. La classe ObjectInputStream

Cette classe permet de déssérialiser un objet.

```
import java.io.*;
public class DeSerializerPersonne {
```

```
public static void main(String argv[]) {
try {
FileInputStream fichier = new FileInputStream("personne.ser");
ObjectInputStream ois = new ObjectInputStream(fichier);
Personne personne = (Personne) ois.readObject();
System.out.println("Personne : ");
System.out.println("nom : "+personne.getNom());
System.out.println("prenom : "+personne.getPrenom());
System.out.println("taille : "+personne.getTaille());
}
catch (java.io.IOException e) {
e.printStackTrace();
}
catch (ClassNotFoundException e) {
e.printStackTrace();
}
}
}
```

Résultat :

```
C:\dej>java DeSerializerPersonne
Personne :
nom : Dupond
prenom : Jean
taille : 175
```

On créer un objet de la classe FileInputStream qui représente le fichier contenant l'objet sérialisé. On créer un objet de type ObjectInputStream en lui passant le fichier en paramètre. Un appèle à la méthode readObject() retourne l'objet avec un type Object. Un cast est nécessaire pour obtenir le type de l'objet. La méthode close() permet de terminer l'opération.
Si la classe a changée entre le moment ou elle a été sérialisée et le moment ou elle est déserialisée, une exception est levée :

```
Exemple : la classe Personne est modifiée et recompilée
C:\temp>java DeSerializerPersonne
java.io.InvalidClassException: Personne; Local class not compatible: stream class
desc serialVersionUID=-2739669178469387642 local class serialVersionUID=39870587
36962107851
atjava.io.ObjectStreamClass.validateLocalClass(ObjectStreamClass.java:4
38)
at java.io.ObjectStreamClass.setClass(ObjectStreamClass.java:482)
at java.io.ObjectInputStream.inputClassDescriptor(ObjectInputStream.java
:785)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:353)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:978)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:369)
at java.io.ObjectInputStream.readObject(ObjectInputStream.java:232)
at DeSerializerPersonne.main(DeSerializerPersonne.java:9)
```

La classe ObjectInputStream possède de la même façon que la classe ObjectOutputStream des méthodes pour lire des données de type primitives : readInt(), readDouble(), readFloat …

Lors de la deserialisation, le constructeur de l'objet n'est jamais utilisé.

2.1.4. Le mot clé transient

Le contenu des attributs sont visibles dans le flux dans lequel est sérialisé l'objet. Il est ainsi possible pour toute personne ayant accès au flux de voir le contenu de chaque attribut même si ceux si sont private. Ceci peut poser des problèmes de sécurité surtout si les données sont sensibles.

Java introduit le mot clé ***transient*** qui précise que l'attribut qu'il qualifie ne doit pas être inclus dans un processus de sérialisation et donc de deserialisation.

```
private transient String codeSecret;
```

Lors de la deserialisation, les champs transient sont initialisés avec la valeur null. Ceci peut poser des problèmes à l'objet qui doit gérer cette état pour éviter d'avoir des exceptions de type NullPointerException.

2.2. Les TP de manipulation de la sérialisation binaire

La sérialisation convertit (sauvegarde) les valeurs stockées dans les attributs d'un objet en flux de données (fichiers).

-Si les fichiers générés par une sérialisation sont binaires, on parle d'une sérialisation binaire.

-Si les fichiers générés par une sérialisation sont en XML, on parle d'une sérialisation XML.

La sérialisation prend son sens dans une communication distribuée (invocation d'une méthode distante). Dans ce cas les objets sérialisés sont :

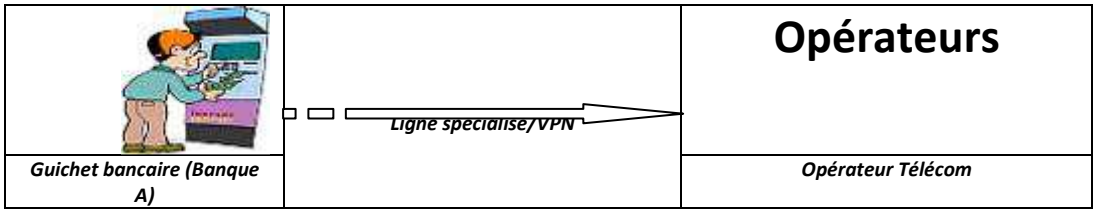
Les objets à passer en arguments pour invoquer la méthode distante.

L'objet qui sera retourné par la méthode distante.

La désérialisation est l'opération inverse de la sérialisation. Elle convertit un flux de données en objets.

Objets	⟶	Sérialisation des objets	⟶	Fichiers
Fichiers	⟶	Désérialisation des fichiers	⟶	Objets

Une classe est dite sérialisable lorsqu'elle implémente l'interface java.io.Serializable.
Dans la suite de cet atelier, nous traiterons le cas du règlement d'une facture d'un téléphone mobile à partir du guichet bancaire automatique (La banque A à titre d'exemple).



Pour régler sa facture, le client doit saisir le numéro de téléphone relatif à la facture et il doit choisir la facture à régler. Supposons alors qu'une facture est définie par son numéro et son client.

2.2.1 Sérialisation binaire des objets de la classe Facture.

La sérialisation des objets de la classe Facture se fait coté guichet bancaire, car c'est à ce niveau que l'opération de règlement est déclenchée.

1. Créer un projet « Guichet Bancaire » et un package « ma.reglement.factures.model»
2. Dans le package « ma.reglement.factures.model» :
   - Créer une classe Client ayant l'attribut numTelephone de type chaine de caractères.
   - Générer automatiquement les Getters et les Setters de l'attribut numTelephone.
   - Surcharger automatiquement le constructeur de la classe Client par l'attribut numTelephone.

```
package ma.reglement.factures.model;

public class Client {

	private String numTelephone;

	public String getNumTelephone() {

		return numTelephone;

	}


	public void setNumTelephone(String numTelephone) {

		this.numTelephone = numTelephone;

	}

	public Client(String numTelephone) {

		this.numTelephone = numTelephone;

	}
```

```
}
```

3. Dans le package « ma.reglement.factures.model» :
   - Créer une classe Facture ayant les deux attributs numFacture de type chaine de caractères et clientFacture de type Client
   - Générer automatiquement les Getters et les Setters des deux attributs numFacture et clientFacture de la classe Facture
   - Surcharger automatiquement le constructeur de la classe Facture par les deux attributs numFacture et clientFacture.

```
package ma.reglement.factures.model;

public class Facture {

	private String numFacture;

	private Client clientFacture;

	public String getNumFacture() {

		return numFacture;

	}

	public void setNumFacture(String numFacture) {

		this.numFacture = numFacture;

	}

	public Client getClientFacture() {

		return clientFacture;

	}

	public Facture(String numFacture, Client clientFacture) {

		super();

		this.numFacture = numFacture;

		this.clientFacture = clientFacture;

	}

	public void setClientFacture(Client clientFacture) {

		this.clientFacture = clientFacture;

	}

}
```

4. Créer un autre package « ma.reglement.factures.main»

5. Dans le package « ma.reglement.factures.main» :
   - Créer une classe SerialisationDuGuichet contenant la méthode main du Java.
   - Dans la méthode main, créer le code Java pour sérialiser les objets de la classe Facture.

```
package ma.reglement.factures.main;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectOutputStream;

import ma.reglement.factures.model.Client;

import ma.reglement.factures.model.Facture;

public class SerialisationDuGuichet {

public static void main(String[] args) {

try {

      Client client = new Client("0661503228");

      Facture facture = new Facture("FAC0001", client);

      FileOutputStream           fos           =           new
FileOutputStream("c:/test.txt");

      ObjectOutputStream os = new ObjectOutputStream(fos);

      os.writeObject(facture);

      } catch (FileNotFoundException e) {

            e.printStackTrace();

      } catch (IOException e) {

            e.printStackTrace();

      }

}}
```

Lors de la sérialisation des objets, le développeur doit gérer les deux exceptions suivantes : `FileNotFoundException` et `IOException`

6. Faire l'exécution du programme de la sérialisation de l'objet Facture. Remarquer l'exception signalant que les objets de la classe Facture ne sont pas sérialisables.

```
java.io.NotSerializableException: ma.reglement.factures.model.Facture
	at java.io.ObjectOutputStream.writeObject0(Unknown Source)
	at java.io.ObjectOutputStream.writeObject(Unknown Source)
	at ma.reglement.factures.main.SerialisationDuGuichet.main(SerialisationDuGuichet.java:22)
```

Seuls les objets des classes qui implémentent l'interface java.io.Serializable peuvent être sauvegardés sous de forme de fichiers.

7. Ajouter le lien d'implémentation entre la classe Facture et l'interface java.io.Serializable et refaire l'exécution du programme de la sérialisation. Remarque que l'exception java.io.NotSerializableException persiste toujours mais cette fois c'est par rapport à la classe Client.

   La classe Facture après ajout de implements Serializable

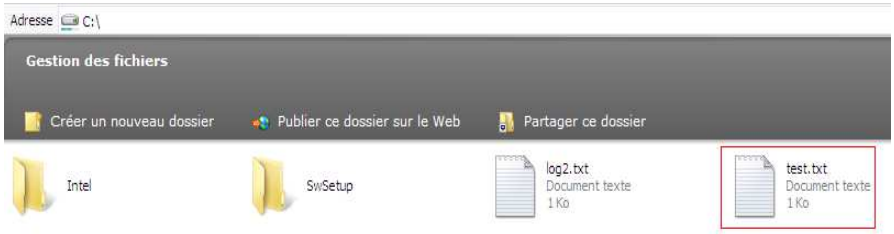
```
package ma.reglement.factures.model;

import java.io.Serializable;

public class Facture implements Serializable{

	//Contenu de la classe Facture

}
```

   L'exception après exécution du programme de la sérialisation.

```
java.io.NotSerializableException: ma.reglement.factures.model.Client
	at java.io.ObjectOutputStream.writeObject0(Unknown Source)
	at java.io.ObjectOutputStream.defaultWriteFields(Unknown Source)
	at java.io.ObjectOutputStream.writeSerialData(Unknown Source)
	at java.io.ObjectOutputStream.writeOrdinaryObject(Unknown Source)
	at java.io.ObjectOutputStream.writeObject0(Unknown Source)
	at java.io.ObjectOutputStream.writeObject(Unknown Source)
	at ma.reglement.factures.main.SerialisationDuGuichet.main(SerialisationDuGuichet.java:22)
```

	*Pour sérialiser un objet, il faut que sa classe implémente l'interface java.io.Serializable et que toutes les classes de ses attributs implémentent également l'interface java.io.Serializable*

8. Maintenant ajouter le lien d'implémentation entre la classe Client et l'interface java.io.Serializable et refaire l'exécution du programme de la sérialisation. Remarquer que l'exception java.io.NotSerializableException ne persiste plus et qu'un fichier est généré dans votre disque dur.



2.2.2 Sérialisation binaire et la relation « avoir a »

Une classe A est en relation « avoir à » avec une autre classe B si parmi les attributs de la classe A on trouve un attribut de type la classe B. (Dans notre exemple la classe Facture est en relation « Avoir à » avec la classe Client)

Nous allons étudier les différents cas relatifs à la sérialisation des objets de la classe Facture sous l'implication suivante :

Sérialiser l'objet Facture ⟹ Sérialiser l'objet Client

2.2.2.1: Le développeur a accès à la classe Client

Le développeur de la classe Facture ***a l'accès*** à la source de la classe Client (Client.java). La solution consiste dans ce cas à s'assurer que Client et Facture implémente l'interface Serializable. (Ce cas est traité dans la première partie de cet atelier)

2.2.2.2 : Le développeur n'a pas l'accès à la source de la classe Client

Le développeur de la classe Facture ***n'a pas l'accès*** à la source de la classe Client et que la classe Client n'implémente pas l'interface Serializable.

La Solution dans ce cas consiste à créer une classe fille de la classe Client (ClientFille) et l'utiliser comme attribut de la classe Facture, la classe ClientFille doit implémenter l'interface Serializable

9. Supprimer le lien d'implémentation avec l'interface Serializable à partir de la classe Client. Exécuter la sérialisation et remarquer l'impossibilité de sérialiser les objets Facture si on n'a pas l'accès à la source de la classe Client.

10. Dans le package « ma.reglement.factures.model», créer la classe ClientFille, classe fille de la classe Client. La classe ClientFille doit implémenter l'interface Serializable. Surcharger le constructeur de la classe ClientFille.

```
package ma.reglement.factures.model;

import java.io.Serializable;

public class ClientFille extends Client implements Serializable
{

	public ClientFille(String numTelephone) {

		super(numTelephone);

	}

}
```

11. Modifier la classe Facture en remplacer le type de l'attribut clientFacture par ClientFille. Régénérer les getters et les setters et surcharger le constructeur de la classe Facture.

```
package ma.reglement.factures.model;

import java.io.Serializable;

public class Facture implements Serializable{

	private String numFacture;

	private ClientFille clientFacture;

	public String getNumFacture() {

		return numFacture;

	}

	public void setNumFacture(String numFacture) {

		this.numFacture = numFacture;

	}

	public ClientFille getClientFacture() {

		return clientFacture;

	}
```

```
	public Facture(String numFacture, ClientFille
clientFacture) {

		super();

		this.numFacture = numFacture;

		this.clientFacture = clientFacture;

	}

	public void setClientFacture(ClientFille clientFacture) {

		this.clientFacture = clientFacture;

	}

}
```

12. Modifier la classe SerialisationDuGuichet en utilisant ClientFille à la place de Client. Faire l'exécution et s'assurer de la réussite de la sérialisation.

```
   public static void main(String[] args) {

	try {

		ClientFille client = new ClientFille("0661503228");

		Facture facture = new Facture("FAC0001", client);

		FileOutputStream fos = new
FileOutputStream("c:/test.txt");

		ObjectOutputStream os = new ObjectOutputStream(fos);

		os.writeObject(facture);

	} catch (FileNotFoundException e) {

		e.printStackTrace();

	} catch (IOException e) {

		e.printStackTrace();

	}

   }
```

2.2.2.3 : Le développeur n'a pas l'accès à la source de la classe Client et que la classe Client est finale

Le développeur qui cherche à sérialiser les objets de la classe Facture n'a pas l'accès à la source de la classe Client et que la classe Client n'implémente pas l'interface Serializable et que la classe Client est finale.

La première solution dans ce cas : Si la classe Client est finale, alors le développeur ne peut plus créer des classes filles comme abordé dans le deuxième cas. Maintenant si le contenu de l'objet clientFacture n'est pas important pour le destinataire, ce dernier peut être ignoré lors de la sérialisation en le déclarant transient.

transient est un mot réservé en java qui permet d'ignorer la sérialisation d'un attribut d'un objet.

13. Mettre la classe Client comme final. Remarque que cette classe ne peut plus être étendue.

```
public final class Client {

	// Le contenu de la classe Client

}
```

14. Supprimer la classe ClientFille et utiliser la classe Client à sa place. Exécuter le programme de la sérialisation et remarquer l'impossibilité de sérialiser.
15. Dans la classe Facture, faire précéder l'attribut clientFacture par transient. Exécuter le programme de la sérialisation et remarquer.

```
package ma.reglement.factures.model;

import java.io.Serializable;

public class Facture implements Serializable{

	private String numFacture;

	private transient Client clientFacture;

	public String getNumFacture() {

		return numFacture;

	}

	public void setNumFacture(String numFacture) {

		this.numFacture = numFacture;

	}

	public Client getClientFacture() {

		return clientFacture;

	}

	public Facture(String numFacture, Client clientFacture) {
```

```
            super();
            this.numFacture = numFacture;
            this.clientFacture = clientFacture;
      }
      public void setClientFacture(Client clientFacture) {
            this.clientFacture = clientFacture;
      }
}
```

16. Créer une classe DeserialiserIam contenant la méthode main et qui permet de dé sérialiser le fichier précédemment crée. S'assurer que la valeur de l'attribut clientFacture est nulle.

```
      public static void main(String[] args) {
            try {
                  FileInputStream fos = new
FileInputStream("c:/test.txt");
                  ObjectInputStream os = new
ObjectInputStream(fos);
                  Facture f = (Facture) os.readObject();
                  System.out.println("La valeur du client est :
"
                              + f.getClientFacture());
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
            } catch (IOException e) {
                  e.printStackTrace();
            } catch (ClassNotFoundException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
```

La deuxième solution dans ce cas : Maintenant si le contenu de l'objet clientFacture est important pour le destinataire, le développeur doit faire une sérialisation

manuelle en introduisant les deux méthodes writeObjet(ObjectOutputStrem os) et readObject(ObjectInputStrem os) au niveau de la classe Facture.

17. Garder l'attribut clientFacture déclaré transient.
18. Créer les deux méthodes writeObjet(ObjectOutputStrem os) et readObject(ObjectInputStrem os) dans la classe Facture comme suit :

```
package ma.reglement.factures.model;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class Facture implements Serializable{

	private String numFacture;

	private transient Client clientFacture;

	public String getNumFacture() {

		return numFacture;

	}

	public void setNumFacture(String numFacture) {

		this.numFacture = numFacture;

	}

	public Client getClientFacture() {

		return clientFacture;

	}

	public Facture(String numFacture, Client clientFacture) {

		super();

		this.numFacture = numFacture;

		this.clientFacture = clientFacture;

	}

	public void setClientFacture(Client clientFacture) {

		this.clientFacture = clientFacture;

	}
```

```
	private void writeObject(ObjectOutputStream os) {
		try {
			os.defaultWriteObject();

	os.writeObject(clientFacture.getNumTelephone());
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	private void readObject(ObjectInputStream is) {
		try {
			is.defaultReadObject();
			clientFacture= new
Client((String)is.readObject());
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch (ClassNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
}
```

Chapitre 3 : L'accès aux bases de données : JDBC

JDBC est l'acronyme de Java DataBase Connectivity et désigne une API définie par Sun pour permettre un accès aux bases de données avec Java.

Ce chapitre présente dans plusieurs sections l'utilisation de cette API

3.1. Présentation des classes de l'API JDBC

Toutes les classes de JDBC sont dans le package java.sql. Il faut donc l'importer dans tous les programmes devant utiliser JDBC.

Il y a 4 classes importantes : DriverManager, Connection, Statement, et ResultSet, chacune correspondante à une étape de l'accès aux données :

DriverManager donne la connexion a la base apres avoir chargé et configurer le driver de la base de données.

Connection réalise la connexion et l'authentification à la base de données.

Statement contient la requête SQL et la transmet à la base de données et permet de faire l'execution.

ResultSet permet de parcourir les informations retournées par la base de données dans le cas d'une sélection de données

Chacune de ces classes dépend de l'instanciation d'un objet de la précédente classe.

3.2. La connexion à une base de données

3.2.1. Le chargement du pilote

Pour se connecter à une base en utilisant un driver spécifique, la documentation du driver fournit le nom de la classe à utiliser. Par exemple, si le nom de la classe est jdbc.DriverXXX, le chargement du driver se fera avec le code suivant : Class.forName("jdbc.DriverXXX");

Exemple : Chargement du pilote pour un base PostgreSQL

```
Class.forName("postgresql.Driver");
```

Il n'est pas nécessaire de créer une instance de cette classe et de l'enregistrer avec le DriverManager car l'appel à Class.forName le fait automatiquement : ce traitement charge le pilote et créer une instance de cette classe.

La méthode static forName() de la classe Class peut lever l'exception java.lang.ClassNotFoundException.

3.2.2. L'établissement de la connection

Pour se connecter à une base de données, il faut instancier un objet de la classe Connection en lui précisant sous forme d'URL la base à accéder.
Exemple : Etablir une connexion sur la base testDB via ODBC

```
String DBurl = "jdbc:odbc:testDB";
```

```
con = DriverManager.getConnection(DBurl);
```

La syntaxe URL peut varier d'un type de base de données à l'autre mais elle est toujours de la forme :

protocole:sous_protocole:nom

« jbdc » désigne le protocole est vaut toujours « jdbc ». « odbc » désigne le sous protocole qui définit le mécanisme de connexion pour un type de bases de données.

Le nom de la base de données doit être celui saisie dans le nom de la source sous ODBC. La méthode getConnection() peut lever une exception de la classe java.sql.SQLException. Le code suivant décrit la création d'une connection avec un user et un mot de passe :

```
Connection con = DriverManager.getConnection(url, "myLogin",
"myPassword");
```

A la place de " myLogin " ; il faut mettre le nom du user qui se connecte à la base et mettre son mot de passe à la place de "myPassword "

3.2.3. L'exécution de requêtes SQL

Les requêtes d'interrogation SQL sont exécutées avec les méthodes d'un objet Statement que l'on obtient à partir d'un objet Connection

```
ResultSet résultats = null;
String requète = "SELECT * FROM client";
try {
Statement stmt = con.createStatement();
résultats = stmt.éxécuteQuery(requète);
} catch (SQLException e) {
//traitement de l'exception
}
```

Un objet de la classe Statement permet d'envoyer des requêtes SQL à la base. La création d'un objet Statement s'effectue à partir d'une instance de la classe Connection

```
Statement stmt = con.createStatement();
```

Pour une requête de type interrogation (SELECT), la méthode à utiliser de la classe Statement est executeQuery. Pour des traitements de mise à jour, il faut utiliser la méthode executeUpdate. Lors de l'appel à la méthode d'exécution, il est nécessaire de lui fournir en paramètre la requête SQL sous forme de chaine.

Le résultat d'une requête d'interrogation est renvoyé dans un objet de la classe ResultSet par la méthode executeQuery().

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employe");
```

La méthode executeUpdate() retourne le nombre d'enregistrement qui ont été mis à jour

```
...
//insertion d'un enregistrement dans la table client
requète = "INSERT INTO client VALUES (3,'client 3','prenom
3')";
try {
Statement stmt = con.createStatement();
int nbMaj = stmt.éxécuteUpdate(requète);
affiche("nb mise a jour = "+nbMaj);
} catch (SQLException e) {
e.printStackTrace();
}
...
```

Lorsque la méthode executeUpdate() est utilisée pour exécuter un traitement de type DDL (Data Defiition Langage : définition de données) comme la création d'un table, elle retourne 0. Si la méthode retourne 0, cela peut signifier deux choses : le traitement de mise à jour n'a affecté aucun enregistrement ou le traitement concernait un traitement de type DDL.

Si l'on utilise executeQuery pour exécuter une requête SQL ne contenant pas d'ordre SELECT, alors une exception de type SQLException est levée.

```
requète = "INSERT INTO client VALUES (4,'client 4','prenom
4')";
try {
Statement stmt = con.createStatement();
ResultSet résultats = stmt.éxécuteQuery(requète);
} catch (SQLException e) {
e.printStackTrace();
}
```

résultat :

```
java.sql.SQLException: No ResultSet was produced
java.lang.Throwable(java.lang.String)
java.lang.Exception(java.lang.String)
java.sql.SQLException(java.lang.String)
java.sql.ResultSet sun.jdbc.odbc.JdbcOdbcStatement
```

Attention : dans ce cas la requête est quand même effectuée. Dans l'exemple, un nouvel enregistrement est créé dans la table. Il n'est pas nécessaire de définir un objet Statement pour chaque ordre SQL : il est possible d'un définir un et de le réutiliser.

3.2.4. La classe ResultSet

C'est une classe qui représente une abstraction d'une table qui se compose de plusieurs enregistrements constitués de colonnes qui contiennent les données.

Les principales méthodes pour obtenir des données sont :
getInt(int) : retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme d'entier.

getInt(String) : retourne le contenu de la colonne dont le nom est passé en paramètre sous forme d'entier.

getFloat(int) : retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de nombre flottant.

getFloat(String) : retourne le contenu de la colonne dont le nom est passé en paramètre sous forme de nombre flottant.

getDate(int) : retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme de date.

getDate(String) : retourne le contenu de la colonne dont le nom est passé en paramètre sous forme de date.

next() : se déplace sur le prochain enregistrement : retourne false si la fin est atteinte Close() ferme le ResultSet

getMetaData() retourne un objet ResultSetMetaData associé au ResultSet.

La méthode getMetaData() retourne un objet de la classe ResultSetMetaData qui permet d'obtenir des informations sur le résultat de la requête. Ainsi, le nombre de colonne peut être obtenu grâce à la méthode getColumnCount de cet objet.

```
ResultSetMetaData rsmd;
rsmd = results.getMetaData();
nbCols = rsmd.getColumnCount();
```

La méthode next() déplace le curseur sur le prochain enregistrement. Le curseur pointe initialement juste avant le premier enregistrement : il est nécessaire de faire un premier appel à la méthode next() pour ce placer sur le premier enregistrement.

Des appels successifs à next permettent de parcourir l'ensemble des enregistrements.
Elle retourne false lorsqu'il n'y a plus d'enregistrement. Il faut toujours protéger le parcours d'une table dans un bloc de capture d'exception

```
//parcours des données retournées
try {
ResultSetMetaData rsmd = résultats.getMetaData();
int nbCols = rsmd.getColumnCount();
boolean encore = résultats.next();
while (encore) {
for (int i = 1; i <= nbCols; i++)
System.out.print(résultats.getString(i) + " ");
```

```
System.out.println();
encore = résultats.next();
}
résultats.close();
} catch (SQLException e) {
//traitement de l'exception
}
```

Les méthodes getXXX() permettent d'extraire les données selon leur type spécifiée par XXX tel que getString(), getDouble(), getInteger(), etc … . Il existe deux formes de ces méthodes : indiquer le numéro la colonne en paramètre (en commençant par 1) ou indiquer le nom de la colonne en paramètre. La première méthode est plus efficace mais peut générer plus d'erreurs à l'exécution notamment si la structure de la table évolue.

Attention : il est important de noter que ce numéro de colonne fourni en paramètre fait référence au numéro de colonne de l'objet resultSet (celui correspondant dans l'ordre SELECT) et non au numéro de colonne de la table.

La méthode getString() permet d'obtenir la valeur d'un champ de n'importe quel type.

3.2.4. Exemple complet de mise à jour et de sélection sur une table

```
import java.sql.*;
public class TestJDBC1 {
private static void affiche(String message) {
System.out.println(message);
}
private static void arret(String message) {
System.err.println(message);
System.exit(99);
}
public static void main(java.lang.String[] args) {
Connection con = null;
ResultSet résultats = null;
String requète = "";
// chargement du pilote
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (ClassNotFoundException e) {
arret("Impossible de charger le pilote jdbc:odbc");
}
//connection a la base de données
affiche("connection a la base de données");
try {
String DBurl = "jdbc:odbc:testDB";
con = DriverManager.getConnection(DBurl);
} catch (SQLException e) {
arret("Connection à la base de données impossible");
}
//insertion d'un enregistrement dans la table client
affiche("creation enregistrement");
requète = "INSERT INTO client VALUES (3,'client 3','client
4')";
```

```
try {
Statement stmt = con.createStatement();
int nbMaj = stmt.éxécuteUpdate(requète);
affiche("nb mise a jour = "+nbMaj);
} catch (SQLException e) {
e.printStackTrace();
}
//creation et execution de la requète
affiche("creation et execution de la requète");
requète = "SELECT * FROM client";
try {
Statement stmt = con.createStatement();
résultats = stmt.éxécuteQuery(requète);
} catch (SQLException e) {
arret("Anomalie lors de l'execution de la requète");
}
//parcours des données retournées
affiche("parcours des données retournées");
try {
ResultSetMetaData rsmd = résultats.getMetaData();
int nbCols = rsmd.getColumnCount();
boolean encore = résultats.next();
while (encore) {
for (int i = 1; i <= nbCols; i++)
System.out.print(résultats.getString(i) + " ");
System.out.println();
encore = résultats.next();
}
résultats.close();
} catch (SQLException e) {
arret(e.getMessage());
}
affiche("fin du programme");
System.exit(0);
}
}
```

Résultat :

```
connection a la base de données
creation enregistrement
nb mise a jour = 1
creation et execution de la requète
parcours des données retournées
1.0 client 1 prenom 1
2.0 client 2 prenom 2
3.0 client 3 client 4
fin du programme
```

Chapitre 4 : Java Native Interface JNI

JNI est l'acronyme de Java Native Interface. C'est une technologie qui permet d'utiliser du code natif, notamment C, dans une classe Java.

L'inconvénient majeur de cette technologie est d'annuler la portabilité du code Java. En contre partie cette technologie peut être très utile dans plusieurs cas :

- Pour des raisons de performance
- Utiliser des composants éprouvés déjà existants

La mise en oeuvre de JNI nécessite plusieurs étapes :

- la déclaration et l'utilisation de la ou des méthodes natives dans la classe Java
- la compilation de la classe Java
- la génération du fichier d'en-tête avec l'outil javah
- l'écriture du code natif en utilisant entre autres les fichiers d'en-tête fournis par le JDK et celui généré précédemment
- la compilation du code natif sous la forme d'une bibliothèque

La bibliothèque est donc dépendante du système d'exploitation pour lequel elle est développée : .dll pour les systèmes de type Windows, .so pour les systèmes de type Unix, ...

Ce chapitre contient plusieurs sections :

4.1 La déclaration et l'utilisation d'une méthode native

La déclaration dans le code source Java est très facile puisqu'il suffit de déclarer la signature de la méthode avec le modificateur native. Le modificateur permet au compilateur de savoir que cette méthode est contenue dans une bibliothèque native.

Il ne doit pas y avoir d'implémentation même pas un corps vide pour une méthode déclarée native.

```
class TestJNI1 {
public native void afficherBonjour();

static {
System.loadLibrary("mabibjni");
}

public static void main(String[] args) {
new TestJNI1().afficherBonjour();
}
}
```

Pour pouvoir utiliser une méthode native, il faut tout d'abord charger la bibliothèque. Pour réaliser ce chargement, il faut utiliser la méthode statique loadLibrary() de la classe system et obligatoirement s'assurer que la bibliothèque est chargée avant le premier appel de la méthode native.

Le plus simple pour assurer ce chargement est de le demander dans un morceau de code d'initialisation statique de la classe.

```
class TestJNI1 {
public native void afficherBonjour();
static {
System.loadLibrary("mabibjni");
}
}
```

Le nom de la bibliothèque fournie en paramètre doit être indépendant de la plate-forme utilisée : il faut préciser le nom de la bibliothèque sans son extension. Le nom sera automatiquement adapté selon le système d'exploitation sur lequel le code Java est exécuté.

L'utilisation de la méthode native dans le code Java se fait de la même façon qu'une méthode classique.

```
class TestJNI1 {
public native void afficherBonjour();
static {
System.loadLibrary("mabibjni");
}
```

```
public static void main(String[] args) {
new TestJNI1().afficherBonjour();
}
}
```

4.2 La génération du fichier d'en-tête

L'outil javah fourni avec le JDK permet de générer un fichier d'en-tête qui va contenir la définition dans le langage C des fonctions correspondant aux méthodes déclarées native dans le code source Java.

Javah utilise le bytecode pour générer le fichier .h. Il faut donc que la classe Java soit préalablement compilée.

La syntaxe est donc : javah -jni nom_fichier_sans_extension

```
01.D:\java\test\jni>dir
02.03/12/2003  14:39       <DIR>          .
03.03/12/2003  14:39       <DIR>          ..
04.03/12/2003  14:39                   230 TestJNI1.java
05.2 fichier(s)              230 octets
06.2 Rép(s)   2 200 772 608 octets libres
07.D:\java\test\jni>javac TestJNI1.java
08.D:\java\test\jni>javah -jni TestJNI1
09.D:\java\test\jni>dir
10.Répertoire de D:\java\test\jni
11.03/12/2003  14:39       <DIR> .
12.03/12/2003  14:39       <DIR> ..
13.03/12/2003  14:39                   459 TestJNI1.class
14.03/12/2003  14:39                   399 TestJNI1.h
15.03/12/2003  14:39                   230 TestJNI1.java
16.3 fichier(s) 1 088 octets
17.2 Rép(s) 2 198 208 512 octets libres

18.D:\java\test\jni>
```

Le fichier TestJNI1.h généré est le suivant :

```
01./* DO NOT EDIT THIS FILE - it is machine generated */
02.#include <jni.h>
03./* Header for class TestJNI1 */
04.
05.#ifndef _Included_TestJNI1
06.#define _Included_TestJNI1
07.#ifdef __cplusplus
08.extern " C " {
09.#endif
10./*
11.*  Class:     TestJNI1
12.*  Method:    afficherBonjour
13.*  Signature: ()V
14.*/
15.JNIEXPORT void JNICALL Java_TestJNI1_afficherBonjour(JNIEnv *, jobject);
```

```
16.#ifdef __cplusplus
17.}
18.#endif

19.#endif
```

Le nom de chaque fonction native respecte le format suivant : Java_nomPleinementQualifieDeLaClasse_NomDeLaMethode

Ce fichier doit être utilisé dans l'implémentation du code de la fonction.

Même si la méthode native est déclarée sans paramètre, il y a toujours deux paramètres passés à la fonction native :

- un pointeur vers une structure JniEnv : cet structure permet d'invoquer certaines fonctionnalités natives de JNI grâce à un tableau de pointeurs de fonctions initialisé par la JVM
- jobject qui est l'objet lui même : c'est l'équivalent du mot clé this dans le code Java

4.3. L'écriture du code natif en C

La bibliothèque contenant la ou les fonctions qui seront appelées doit être écrite dans un langage (c ou c++) et compilée.

Pour l'écriture en C, facilitée par la génération du fichier.h, il est nécessaire en plus des includes liées au code des fonctions d'inclure deux fichiers d'en-tête :

- jni.h qui est fourni avec le JDK
- le fichier .h généré par la commande javah

```
01.#include <jni.h>
02.#include <stdio.h>
03.#include "TestJNI1.h"
04.
05.JNIEXPORT void JNICALL
06.Java_TestJNI1_afficherBonjour(JNIEnv *env, jobject obj)
07.{
08.printf(" Bonjour\n ");
09.return;
10.}
```

Il faut compiler ce fichier source sous la forme d'un fichier objet .o

```
1.D:\java\test\jni>gcc -c -I"C:\j2sdk1.4.2_02\include" -
I"C:\j2sdk1.4.2_02\include
2.\win32" -o TestJNI.o TestJNI.c
```

Il faut ensuite définir un fichier .def qui contient la définition des fonctions exportées par la bibliothèque

```
1.EXPORTS
2.Java_TestJNI1_afficherBonjour
```

Il ne reste plus qu'à générer la dll.

```
01.D:\java\test\jni>gcc -shared -o mabibjni.dll TestJNI.o TestJNI.def
02.Warning: resolving _Java_TestJNI1_afficherBonjour by linking to
_Java_TestJNI1_a
03.fficherBonjour@8
04.Use-enable-stdcall-fixup to disable these warnings
05.Use-disable-stdcall-fixup to disable these fixups
06.
07.D:\java\test\jni>dir
08.Répertoire de D:\java\test\jni
09.03/12/2003  16:22       <DIR>          .
10.03/12/2003  16:22       <DIR>          ..
11.03/12/2003  16:22               12 017 mabibjni.dll
12.03/12/2003  15:58                  193 TestJNI.c
13.03/12/2003  16:20                   40 TestJNI.def
14.03/12/2003  16:04                  543 TestJNI.o
15.03/12/2003  14:39                  459 TestJNI1.class
16.03/12/2003  14:39                  399 TestJNI1.h
17.03/12/2003  14:39                  230 TestJNI1.java
18.9 fichier(s)           14 074 octets
19.2 Rép(s)   2 198 392 832 octets libres
20.
21.D:\java\test\jni>
```

Il ne reste plus qu'à exécuter le code Java dans une machine virtuelle.

```
1.D:\java\test\jni>java TestJNI1
2.Bonjour
3.D:\java\test\jni>
```

Il est intéressant de noter que tant que la signature de la méthode native ne change pas, il est inutile de recompiler la classe Java si la fonction dans la bibliothèque est modifiée et recompilée.